Contents

- 1 Curie's advanced usage manual
- 2 Optimization
  - 2.1 Compilation options
    - 2.1.1 Intel
      - 2.1.1.1 Intel Sandy Bridge processors
    - 2.1.2 GNU
- 3 Submission
  - 3.1 Choosing or excluding nodes
- 4 MPI
  - 4.1 Embarrassingly parallel jobs and MPMD jobs
  - 4.2 BullxMPI
    - 4.2.1 MPMD jobs
    - 4.2.2 Tuning BullxMPI
    - 4.2.3 Optimizing with BullxMPI
    - 4.2.4 Debugging with BullxMPI
- 5 Process distribution, affinity and binding
  - 5.1 Introduction
    - 5.1.1 Hardware topology
    - 5.1.2 Definitions
    - 5.1.3 Process distribution
    - 5.1.4 Why is affinity important for improving performance ?
    - 5.1.5 CPU affinity mask
  - 5.2 SLURM
    - 5.2.1 Process distribution
      - 5.2.1.1 Curie hybrid node
    - 5.2.2 Process binding
  - 5.3 BullxMPI
    - 5.3.1 Process distribution
    - 5.3.2 Process binding
    - 5.3.3 Manual process management
- 6 Using GPU
  - 6.1 Two sequential GPU runs on a single hybrid node
- 7 Profiling
  - 7.1 PAPI
  - 7.2 VampirTrace/Vampir
    - 7.2.1 Basics
    - 7.2.2 Tips
    - 7.2.3 Vampirserver
    - 7.2.4 CUDA profiling
  - 7.3 Scalasca
    - 7.3.1 Standard utilization
    - 7.3.2 Scalasca + Vampir
    - 7.3.3 Scalasca + PAPI
  - 7.4 Paraver
    - 7.4.1 Trace generation
    - 7.4.2 Converting traces to Paraver format
    - 7.4.3 Launching Paraver

Curie's advanced usage manual

If you have suggestions or remarks, please contact us : hotline.tgcc@cea.fr

Optimization

Compilation options

Compilers provides many options to optimize a code. These options are described in the following section.

Intel

- -opt_report : generates a report which describes the optimisation in stderr (-O3 required)
- -ip, -ipo : inter-procedural optimizations (mono and multi files). The command xiar must be used instead of ar to generate a static library file with objects compiled with -ipo option.
- -fast : default high optimisation level (-O3 -ipo -static). + Carefull : This option is not allowed using MPI, the MPI context needs to call some libraries which only exists in dynamic mode. This is incompatible with the -static option. You need to replace -fast by -O3 -ipo
- -ftz : considers all the denormalized numbers (like INF or NAN) as zeros at runtime.
- -fp-relaxed : mathematical optimisation functions. Leads to a small loss of accuracy.
- -pad : makes the modification of the memory positions operational (ifort only)

There are some options which allow to use specific instructions of Intel processors in order to optimize the code. These options are compatible with most of Intel processors. The compiler will try to generate these instructions if the processor allow it.

- -xSSE4.2 : May generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions.
- -xSSE4.1 : May generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. May generate Intel® SSSE3, SSE3, SSE2, and SSE instructions.
- -xSSSE3 : May generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors.
- -xSSE3 : May generate Intel® SSE3, SSE2, and SSE instructions for Intel processors.
- -xSSE2 : May generate Intel® SSE2 and SSE instructions for Intel processors.
- -xHost : this option will apply one of the previous options depending on the processor where the compilation is performed. This option is recommended for optimizing your code.

None of these options are used by default. The SSE instructions use the vectorization capability of Intel processors.

Intel Sandy Bridge processors

Curie thin nodes use the last Intel processors based on Sandy Bridge architecture. This architecture provides new vectorization instructions called AVX for Advanced Vector eXtensions. The option **-xAVX** allows to generate a specific code for Curie thin nodes.

Be careful, a code generated with -xAVX option runs only on Intel Sandy Bridge processors. Otherwise, you will get this error message:

```
Fatal Error: This program was not built to run in your system.
Please verify that both the operating system and the processor support Intel(R) AVX.
```

Curie login nodes are Curie large nodes with Nehalem-EX processors. AVX codes can be generated on these nodes through cross-compilation by adding -xAVX option. On Curie large node, the -xHost option will not generate a AVX code. If you need to compile with -xHost or if the installation requires some tests (like autotools/configure), you can submit a job which will compile on the Curie thin nodes.

GNU

There are some options which allow usage of specific set of instructions for Intel processors, in order to optimize code behavior. These options are compatible with most of Intel processors. The compiler will try to use these instructions if the processor allow it.

- -mmmx / -mno-mmx : Switch on or off the usage of said instruction set.
- -msse / -mno-sse : idem.
- -msse2 / -mno-sse2 : idem.
- -msse3 / -mno-sse3 : idem.
- -mssse3 / -mno-ssse3 : idem.
- -msse4.1 / -mno-sse4.1 : idem.
- -msse4.2 / -mno-sse4.2 : idem.
- -msse4 / -mno-sse4 : idem.
- -mavx / -mno-avx : idem, **for Curie Thin nodes partition only**.

Submission

Choosing or excluding nodes

SLURM provides the possibility to choose or exclude any nodes in the reservation for your job.

To choose nodes:

```
#!/bin/bash
#MSUB -r MyJob_Para              # Request name
#MSUB -n 32                      # Number of tasks to use
#MSUB -T 1800                    # Elapsed time limit in seconds
#MSUB -o example_%I.o            # Standard output. %I is the jobid
#MSUB -e example_%I.e            # Error output. %I is the jobid
#MSUB -A paxxxx                  # Project ID
#MSUB -E '-w curie[1000-1003]'   # Include 4 nodes (curie1000 to curie1003)

set -x
cd ${BRIDGE_MSUB_PWD}
ccc_mprun ./a.out
```

To exclude nodes:

```
#!/bin/bash
#MSUB -r MyJob_Para              # Request name
#MSUB -n 32                      # Number of tasks to use
#MSUB -T 1800                    # Elapsed time limit in seconds
#MSUB -o example_%I.o            # Standard output. %I is the jobid
#MSUB -e example_%I.e            # Error output. %I is the jobid
#MSUB -A paxxxx                  # Project ID
#MSUB -E '-x curie[1000-1003]'   # Exclude 4 nodes (curie1000 to curie1003)

set -x
cd ${BRIDGE_MSUB_PWD}
ccc_mprun ./a.out
```

MPI

Embarrassingly parallel jobs and MPMD jobs

- An embarrassingly parallel job is a job which launch independent processes. These processes need few or no communications
- A MPMD job is a parallel job which launch different executables over the processes. A MPMD job can be parallel with MPI and can do many communications.

These two concepts are separate but we present them together because the way to launch them on Curie is similar. An simple example in the Curie info page was already given.

In the following example, we use *ccc_mprun* to launch the job. *srun* can be used too. We want to launch *bin0* on the MPI rank 0, *bin1* on the MPI rank 1 and *bin2* on the MPI rank 2. We have first to write a shell script which describes the topology of our job:

launch_exe.sh:

```
#!/bin/bash

if [ $SLURM_PROCID -eq 0 ]
 then
    ./bin0
fi
if [ $SLURM_PROCID -eq 1 ]
 then
    ./bin1
fi
if [ $SLURM_PROCID -eq 2 ]
 then
    ./bin2
fi
```

We can then launch our job with 3 processes:

```
ccc_mprun -n 3 ./launch_exe.sh
```

The script *launch_exe.sh* must have execute permission. When *ccc_mprun* launches the job, it will initialize some environment variables. Among them, *SLURM_PROCID* defines the current MPI rank.

BullxMPI

MPMD jobs

BullxMPI (or OpenMPI) jobs can be launched with *mpirun* launcher. In this case, we have other ways to launch MPMD jobs (see embarrassingly parallel jobs section).

We take the same example in the embarrassingly parallel jobs section. There are then two ways for launching MPMD scripts

- We don't need the *launch_exe.sh* anymore. We can launch directly the job with *mpirun* command:

```
mpirun -np 1 ./bin0 : -np 1 ./bin1 : -np 1 ./bin2
```

- In the *launch_exe.sh*, we can replace *SLURM_PROCID* by *OMPI_COMM_WORLD_RANK*:

launch_exe.sh:

```
#!/bin/bash

if [ ${OMPI_COMM_WORLD_RANK} -eq 0 ]
 then
    ./bin0
fi
if [ ${OMPI_COMM_WORLD_RANK} -eq 1 ]
 then
    ./bin1
fi
if [ ${OMPI_COMM_WORLD_RANK} -eq 2 ]
 then
    ./bin2
fi
```

We can then launch our job with 3 processes:

```
mpirun -np 3 ./launch_exe.sh
```

Tuning BullxMPI

BullxMPI is based on OpenMPI. It can be tuned with parameters. The command *ompi_info -a* gives you a list of all parameters and their descriptions.

```
curie50$ ompi_info -a
(...)
                MCA mpi: parameter "mpi_show_mca_params" (current value: <none>, data source: default value)
                         Whether to show all MCA parameter values during MPI_INIT or not (good for reproducability of MPI jobs for debug purposes). Accepted values are all, default, file, api, and environment
                         - or a comma delimited combination of them
(...)
```

Theses parameters can be modified with environment variables set before the *ccc_mprun* command. The form of the corresponding environment variable is *OMPI_MCA_xxxxx* where *xxxxx* is the parameter.

```
#!/bin/bash
#MSUB -r MyJob_Para             # Request name
#MSUB -n 32                     # Number of tasks to use
#MSUB -T 1800                   # Elapsed time limit in seconds
#MSUB -o example_%I.o           # Standard output. %I is the jobid
#MSUB -e example_%I.e           # Error output. %I is the jobid
#MSUB -A paxxxx                 # Project ID

set -x
cd ${BRIDGE_MSUB_PWD}
export OMPI_MCA_mpi_show_mca_params=all
ccc_mprun ./a.out
```

Optimizing with BullxMPI

You can try theses parameters in order to optimize BullxMPI:

```
export OMPI_MCA_mpi_leave_pinned=1
```

This setting improves the bandwidth for communication if the code uses the same buffers for communication during the execution.

```
export OMPI_MCA_btl_openib_use_eager_rdma=1
```

This parameter optimizes the latence for short messages on Infiniband network. But the code will use more memory.

Be careful, theses parameters are not set by default. They can have influences on the behaviour of your codes.

Debugging with BullxMPI

Sometimes, BullxMPI codes can hang in any collective communication for large jobs. If you find yourself in this case, you can try this parameter:

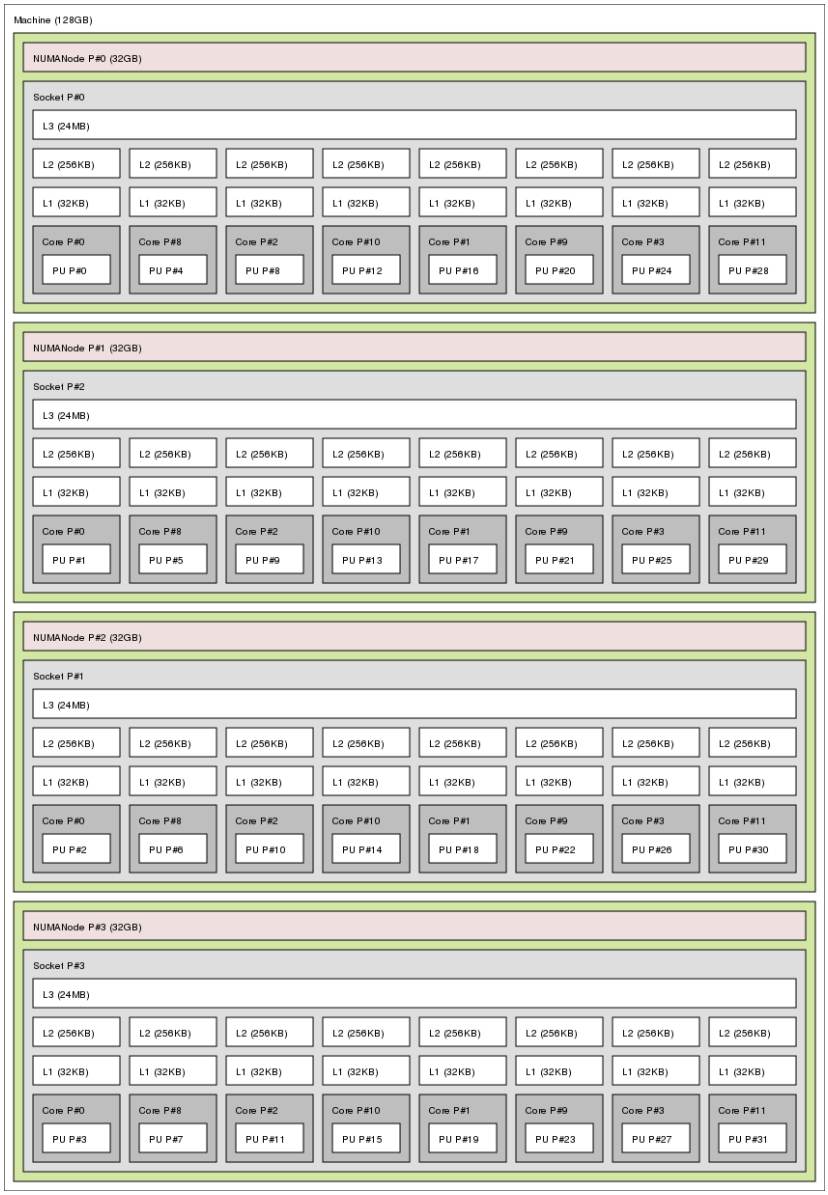
```
export OMPI_MCA_coll="^ghc,tuned"
```

This setting disables optimized collective communications: it can slow down your code if it uses many collective operations.

Process distribution, affinity and binding

Introduction

Hardware topology



Hardware topology of a Curie fat node

The hardware topology is the organization of cores, processors, sockets and memory in a node. The previous image was created with *hwloc*. You can have access to *hwloc* on Curie with the command *module load hwloc*.

Definitions

We define here some vocabulary:

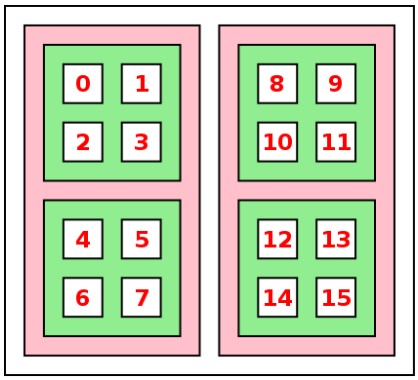
- **Binding** : a Linux process can be bound (or stuck) to one or many cores. It means a process and its threads can run only on a given selection of cores. For example, a process which is bound to a socket on a Curie fat node can run on any of the 8 cores of a processor.
- **Affinity** : it represents the policy of resources management (cores and memory) for processes.
- **Distribution** : the distribution of MPI processes describes how theses processes are spread accross the core, sockets or nodes.

On Curie, the default behaviour for distribution, affinity and binding are managed by SLURM, precisely the *ccc_mprun* command.

Process distribution

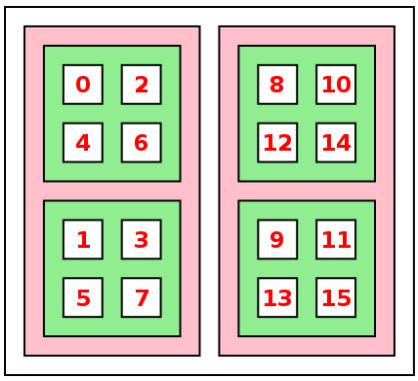
We present here some example of MPI processes distributions.

- **block** or **round** : this is the standard distribution. From SLURM manpage: The block distribution method will distribute tasks to a node such that consecutive tasks share a node. For example, consider an allocation of two nodes each with 8 cores. A block distribution request will distribute those tasks to the nodes with tasks 0 to 7 on the first node, task 8 to 15 on the second node.



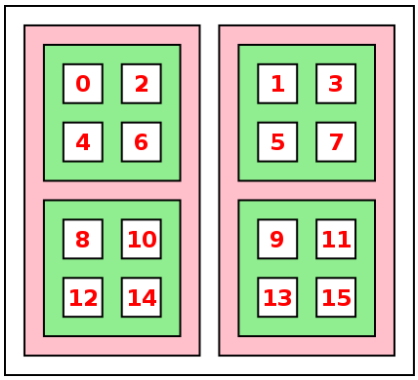
Block distribution by core

- **cyclic** by socket: from SLURM manpage, the cyclic distribution method will distribute tasks to a socket such that consecutive tasks are distributed over consecutive socket (in a round-robin fashion). For example, consider an allocation of two nodes each with 2 sockets each with 4 cores. A cyclic distribution by socket request will distribute those tasks to the socket with tasks 0,2,4,6 on the first socket, task 1,3,5,7 on the second socket. In the following image, the distribution is cyclic by socket and block by node.



Cyclic distribution by socket

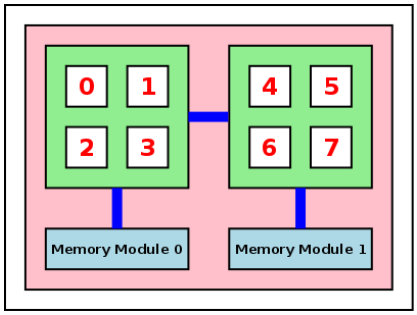
- **cyclic** by node: from SLURM manpage, the cyclic distribution method will distribute tasks to a node such that consecutive tasks are distributed over consecutive nodes (in a round-robin fashion). For example, consider an allocation of two nodes each with 2 sockets each with 4 cores. A cyclic distribution by node request will distribute those tasks to the nodes with tasks 0,2,4,6,8,10,12,14 on the first node, task 1,3,5,7,9,11,13,15 on the second node. In the following image, the distribution is cyclic by node and block by socket.



Block distribution by node

Why is affinity important for improving performance ?

Curie nodes are NUMA (Non-Uniform Memory Access) nodes. It means that it will take longer to access some regions of memory than others. This is due to the fact that all memory regions are not physically on the same bus.

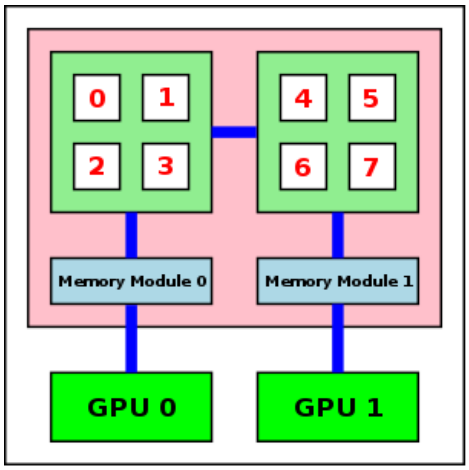


NUMA node : Curie hybrid node

In this picture, we can see that if a data is in the memory module 0, a process running on the second socket like the 4th process will take more time to access the data. We can introduce the notion of *local data* vs *remote data*. In our example, if we consider a process running on the socket 0, a data is *local* if it is on the memory module 0. The data is *remote* if it is on the memory module 1.

We can then deduce the reasons why tuning the process affinity is important:

- Data locality improve performance. If your code use shared memory (like pthreads or OpenMP), the best choice is to regroup your threads on the same socket. The shared datas should be local to the socket and moreover, the datas will potentially stay on the processor's cache.
- System processes can interrupt your process running on a core. If your process is not bound to a core or to a socket, it can be moved to another core or to another socket. In this case, all datas for this process have to be moved with the process too and it can take some time.
- MPI communications are faster between processes which are on the same socket. If you know that two processes have many communications, you can bind them to the same socket.
- On Curie hybrid nodes, the GPUs are connected to buses which are local to socket. Processes can take longer time to access a GPU which is not connected to its socket.



NUMA node : Curie hybrid node with GPU

For all theses reasons, it is better to know the NUMA configuration of Curie nodes (fat, hybrid and thin). In the following section, we will present some ways to tune your processes affinity for your jobs.

CPU affinity mask

The affinity of a process is defined by a mask. A mask is a binary value which length is defined by the number of cores available on a node. By example, Curie hybrid nodes have 8 cores: the binary mask value will have 8 figures. Each figures will have 0 or 1. The process will run only on the core which have 1 as value. A binary mask must be read from right to left.

For example, a process which runs on the cores 0,4,6 and 7 will have as affinity binary mask: *11010001*

SLURM and BullxMPI use theses masks but converted in hexadecimal number.

- To convert a binary value to hexadecimal:

```
$ echo "ibase=2;obase=16;11010001" | bc
D1
```

- To convert a hexadecimal value to binary:

```
$ echo "ibase=16;obase=2;D1" | bc
11010001
```

The numbering of the cores is the PU number from the output of hwloc.

SLURM

SLURM is the default launcher for jobs on Curie. SLURM manages the processes even for sequential jobs. We recommend you to use *ccc_mprun*. By default, SLURM binds processes to a core. The distribution is block by node and by core.

The option *-E* '*--cpu_bind=verbose*' for *ccc_mprun* gives you a report about the binding of processes before the run:

```
$ ccc_mprun -E '--cpu_bind=verbose' -q hybrid -n 8 ./a.out
cpu_bind=MASK - curie7054, task  3  3 [3534]: mask 0x8 set
cpu_bind=MASK - curie7054, task  0  0 [3531]: mask 0x1 set
cpu_bind=MASK - curie7054, task  1  1 [3532]: mask 0x2 set
cpu_bind=MASK - curie7054, task  2  2 [3533]: mask 0x4 set
cpu_bind=MASK - curie7054, task  4  4 [3535]: mask 0x10 set
cpu_bind=MASK - curie7054, task  5  5 [3536]: mask 0x20 set
cpu_bind=MASK - curie7054, task  7  7 [3538]: mask 0x80 set
cpu_bind=MASK - curie7054, task  6  6 [3537]: mask 0x40 set
```

In this example, we can see the process 5 has 20 as hexadecimal mask or 00100000 as binary mask: the 5th process will run only on the core 5.

Process distribution

To change the default distribution of processes, you can use the option *-E* '*-m*' for *ccc_mprun*. With SLURM, you have two levels for process distribution: node and socket.

- Node block distribution:

```
ccc_mprun -E '-m block' ./a.out
```

- Node cyclic distribution:

```
ccc_mprun -E '-m cyclic' ./a.out
```

By default, the distribution over the socket is block. In the following examples for socket distribution, the node distribution will be block.

- Socket block distribution:

```
ccc_mprun -E '-m block:block' ./a.out
```

- Socket cyclic distribution:

```
ccc_mprun -E '-m block:cyclic' ./a.out
```

Curie hybrid node

On Curie hybrid node, each GPU is connected to a socket (see previous picture). It will take longer for a process to access a GPU if this process is not on the same socket of the GPU. By default, the distribution is block by core. Then the MPI rank 0 is located on the first socket and the MPI rank 1 is on the first socket too. The majority of GPU codes will assign GPU 0 to MPI rank 0 and GPU 1 to MPI rank 1. In this case, the bandwidth between MPI rank 1 and GPU 1 is not optimal.

If your code does this, in order to obtain the best performance, you should :

- use the block:cyclic distribution
- if you intend to use only 2 MPI processes per node, you can reserve 4 cores per process with the directive *#MSUB -c 4*. The two processes will be placed on two different sockets.

Process binding

By default, processes are bound to the core. For multi-threaded jobs, processes creates threads: these threads will be bound to the assigned core. To allow these threads to use other cores, SLURM provides the option *-c* to assign many cores to a process.

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 8                   # Number of tasks to use
#MSUB -c 4                   # Assign 4 cores per process
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
```

```
export OMP_NUM_THREADS=4
ccc_mprun ./a.out
```

In this example, our hybrid OpenMP/MPI code runs on 8 MPI processes and each process will use 4 OpenMP threads. We give here an example for the output with the verbose option for binding:

```
$ ccc_mprun ./a.out
cpu_bind=MASK - curie1139, task  5  5 [18761]: mask 0x40404040 set
cpu_bind=MASK - curie1139, task  0  0 [18756]: mask 0x10101010 set
cpu_bind=MASK - curie1139, task  1  1 [18757]: mask 0x10101010 set
cpu_bind=MASK - curie1139, task  6  6 [18762]: mask 0x80808080 set
cpu_bind=MASK - curie1139, task  4  4 [18760]: mask 0x40404040 set
cpu_bind=MASK - curie1139, task  3  3 [18759]: mask 0x20202020 set
cpu_bind=MASK - curie1139, task  2  2 [18758]: mask 0x20202020 set
cpu_bind=MASK - curie1139, task  7  7 [18763]: mask 0x80808080 set
```

We can see here the MPI rank 0 process is launched over the cores 0,8,16 and 24 of the node. These cores are all located on the node's first socket.

Remark: With the *-c* option, SLURM will try to gather at best the cores to have best performances. In the previous example, all the cores of a MPI process will be located on the same socket.

Another example:

```
$ ccc_mprun -n 1 -c 32 -E '--cpu_bind=verbose' ./a.out
cpu_bind=MASK - curie1017, task  0  0 [347710]: mask 0xffffffff set
```

We can see the process is not bound to a core and can run over all cores of a node.

BullxMPI

BullxMPI has its own process management policy. To use it, you have first to disable SLURM's process management policy by adding the directive *#MSUB -E '--cpu_bind=none'* . You can then use BullxMPI launcher *mpirun*:

```
#!/bin/bash
#MSUB -r MyJob_Para              # Request name
#MSUB -n 32                      # Number of tasks to use
#MSUB -x                         # Require a exclusive node
#MSUB -T 1800                    # Elapsed time limit in seconds
#MSUB -o example_%I.o            # Standard output. %I is the jobid
#MSUB -A paxxxx                  # Project ID
#MSUB -E '--cpu_bind=none'       # Disable default SLURM binding

mpirun -np 32 ./a.out
```

Note: In this example, BullxMPI process management policy can be effective only on the 32 cores allocated by SLURM.

The default BullxMPI process management policy is:

- the processes are not bound
- the processes can run on all cores
- the default distribution is block by core and by node

The option *--report-bindings* gives you a report about the binding of processes before the run:

```
#!/bin/bash
#MSUB -r MyJob_Para              # Request name
#MSUB -n 32                      # Number of tasks to use
#MSUB -x                         # Require a exclusive node
#MSUB -T 1800                    # Elapsed time limit in seconds
#MSUB -o example_%I.o            # Standard output. %I is the jobid
#MSUB -A paxxxx                  # Project ID
#MSUB -E '--cpu_bind=none'       # Disable default SLURM binding

mpirun --report-bindings --bind-to-socket --cpus-per-proc 4 -np 8 ./a.out
```

And there is the output:

```
+ mpirun --bind-to-socket --cpus-per-proc 4 -np 8 ./a.out
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],3] to socket 1 cpus 22222222
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],4] to socket 2 cpus 44444444
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],5] to socket 2 cpus 44444444
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],6] to socket 3 cpus 88888888
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],7] to socket 3 cpus 88888888
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],0] to socket 0 cpus 11111111
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],1] to socket 0 cpus 11111111
[curie1342:19946] [[40080,0],0] odls:default:fork binding child [[40080,1],2] to socket 1 cpus 22222222
```

In the following paragraphs, we present the different possibilities of process distribution and binding. These options can be mixed (if possible).

Remark: the following examples use a whole Curie fat node. We reserve 32 cores with *#MSUB -n 32* and *#MSUB -x* to have all the cores and to do what we want with them. This is only examples for simple cases. In others case, there may be conflicts with SLURM.

Process distribution

Block distribution by core:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bycore -np 32 ./a.out
```

Cyclic distribution by socket:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bysocket -np 32 ./a.out
```

Cyclic distribution by node:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -N 16
#MSUB -x                     # Require exclusive nodes
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bynode -np 32 ./a.out
```

Process binding

No binding:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bind-to-none -np 32 ./a.out
```

Core binding:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bind-to-core -np 32 ./a.out
```

Socket binding (the process and his threads can run on all cores of a socket):

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bind-to-socket -np 32 ./a.out
```

You can specify the number of cores to assign to a MPI process:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

mpirun --bind-to-socket --cpus-per-proc 4 -np 8 ./a.out
```

Here we assign 4 cores per MPI process.

Manual process management

BullxMPI gives the possibility to manually assign your processes through a hostfile and a rankfile. An example:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -x                     # Require a exclusive node
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -A paxxxx              # Project ID
#MSUB -E '--cpu_bind=none'   # Disable default SLURM binding

hostname > hostfile.txt
```

```
echo "rank 0=${HOSTNAME} slot=0,1,2,3     " >  rankfile.txt
echo "rank 1=${HOSTNAME} slot=8,10,12,14  " >> rankfile.txt
echo "rank 2=${HOSTNAME} slot=16,17,22,23" >> rankfile.txt
echo "rank 3=${HOSTNAME} slot=19,20,21,31" >> rankfile.txt
mpirun --hostfile hostfile.txt --rankfile rankfile.txt -np 4 ./a.out
```

In this example, there are many steps :

- You have to create a *hostfile* here hostfile.txt where you put the hostname of all nodes your run will use
- You have to create a *rankfile* here rankfile.txt where you assign to each MPI rank the core where it can run. In our example, the process of rank 0 will have as affinity the core 0,1,2 and 3, etc... Be careful, the numbering of the core is different than the hwloc output: on Curie fat node, the eight first core are on the first socket 0, etc...
- you can launch *mpirun* by specifying the hostfile and the rankfile.

Using GPU

Two sequential GPU runs on a single hybrid node

To launch two separate sequential GPU runs on a single hybrid node, you have to set the environment variable CUDA_VISIBLE_DEVICES which enables GPUs wanted. First, create a script to launch binaries:

```
$ cat launch_exe.sh

#!/bin/bash
set -x

export CUDA_VISIBLE_DEVICES=${SLURM_PROCID} # the first process will see only the first GPU and the second process will see only the second GPU.

if [ $SLURM_PROCID -eq 0 ]
then
    ./bin_1 > job_${SLURM_PROCID}.out
fi
if [ $SLURM_PROCID -eq 1 ]
then
    ./bin_2 > job_${SLURM_PROCID}.out
fi
```

/!\ To work correctly, the two binaries have to been sequential (not using MPI).

Then run your script, making sure to submit two MPI processes with 4 cores per process:

```
$ cat multi_jobs_gpu.sh
#!/bin/bash
#MSUB -r jobs_gpu
#MSUB -n 2                # 2 tasks
#MSUB -N 1                # 1 node
#MSUB -c 4                # each task takes 4 cores
#MSUB -q hybrid
#MSUB -T 1800
#MSUB -o multi_jobs_gpu_%I.out
#MSUB -e multi_jobs_gpu_%I.out

set -x
cd $BRIDGE_MSUB_PWD
export OMP_NUM_THREADS=4

ccc_mprun -E '--wait=0' -n 2 -c 4 ./launch_exe.sh
# -E '--wait=0' specify to slurm to not kill the job if one of the two processes is terminated and not the second
```

So your first process will be located on the first CPU socket and the second process will be on the second CPU socket (each socket is linked with a GPU).

```
$ ccc_msub multi_jobs_gpu.sh
```

Profiling

PAPI

PAPI is an API which allows you to retrieve hardware counters from the CPU. Here an example in Fortran to get the number of floating point operations of a matrix DAXPY:

```
program main
 implicit none
 include 'f90papi.h'
 !
 integer, parameter :: size = 1000
 integer, parameter :: ntimes = 10
 double precision, dimension(size,size) :: A,B,C
 integer :: i,j,n
 ! Variable PAPI
 integer, parameter :: max_event = 1
 integer, dimension(max_event) :: event
 integer :: num_events, retval
 integer(kind=8), dimension(max_event) :: values
 ! Init PAPI
 call PAPIf_num_counters( num_events )
 print *, 'Number of hardware counters supported: ', num_events
 call PAPIf_query_event(PAPI_FP_INS, retval)
 if (retval .NE. PAPI_OK) then
    event(1) = PAPI_TOT_INS
 else
    ! Total floating point operations
    event(1) = PAPI_FP_INS
 endif
 ! Init Matrix
 do i=1,size
    do j=1,size
       C(i,j) = real(i+j,8)
       B(i,j) = -i+0.1*j
    enddo
```

```
end do
! Set up counters
num_events = 1
call PAPIf_start_counters( event, num_events, retval)
! Clear the counter values
call PAPIf_read_counters(values, num_events,retval)
! DAXPY
do n=1,ntimes
   do i=1,size
      do j=1,size
        A(i,j) = 2.0*B(i,j) + C(i,j)
      end do
   end do
end do
! Stop the counters and put the results in the array values
call PAPIf_stop_counters(values,num_events,retval)
! Print results
if (event(1) .EQ. PAPI_TOT_INS ) then
   print *, 'TOT Instructions:  ',values(1)
else
   print *, 'FP Instructions:  ',values(1)
end if
end program main
```

To compile, you have to load the PAPI module :

```
bash-4.00$ module load papi/4.1.3
bash-4.00$ ifort -I${PAPI_INC_DIR} papi.f90 ${PAPI_LIBS}
bash-4.00$ ./a.out
Number of hardware counters supported:          7
FP Instructions:                10046163
```

To get the available hardware counters, you can type "papi_avail" commande.

This library can retrieve the MFLOPS of a certain region of your code:

```
program main
implicit none
include 'f90papi.h'
!
integer, parameter :: size = 1000
integer, parameter :: ntimes = 100
double precision, dimension(size,size) :: A,B,C
integer :: i,j,n
! Variable PAPI
integer :: retval
real(kind=4) :: proc_time, mflops, real_time
integer(kind=8) ::  flpins
! Init PAPI
retval = PAPI_VER_CURRENT
call PAPIf_library_init(retval)
if ( retval.NE.PAPI_VER_CURRENT) then
   print*, 'PAPI_library_init', retval
end if
call PAPIf_query_event(PAPI_FP_INS, retval)
! Init Matrix
do i=1,size
   do j=1,size
      C(i,j) = real(i+j,8)
      B(i,j) = -i+0.1*j
   end do
end do
! Setup Counter
call PAPIf_flips( real_time, proc_time, flpins, mflops, retval )
! DAXPY
do n=1,ntimes
   do i=1,size
      do j=1,size
        A(i,j) = 2.0*B(i,j) + C(i,j)
      end do
   end do
end do
! Collect the data into the Variables passed in
call PAPIf_flips( real_time, proc_time, flpins, mflops, retval)
! Print results
print *, 'Real_time: ', real_time
print *, 'Proc_time: ', proc_time
print *, 'Total flpins: ', flpins
print *, 'MFLOPS: ', mflops
!
end program main
```

and the output:

```
bash-4.00$ module load papi/4.1.3
bash-4.00$ ifort -I${PAPI_INC_DIR} papi_flops.f90 ${PAPI_LIBS}
bash-4.00$ ./a.out
Real_time:    6.12500001E-02
 Proc_time:    5.14475589E-02
 Total flpins:                100056592
 MFLOPS:     1944.826
```

If you want more precisions, you can contact us or visit PAPI website.

VampirTrace/Vampir

VampirTrace is a library which let you profile your parallel code by taking traces during the execution of the program. We present here an introduction of Vampir/Vampirtrace.

Basics

First, you must compile your code with VampirTrace compilers. In order to use VampirTrace, you need to load the vampirtrace module:

```
bash-4.00$ module load vampirtrace
bash-4.00$ vtcc -c prog.c
bash-4.00$ vtcc -o prog.exe prog.o
```

Available compilers are :

- *vtcc* : C compiler
- *vtc++*, *vtCC* et *vtcxx* : C++ compilers
- *vtf77* et *vtf90* : Fortran compilers

To compile a MPI code, you should type :

```
bash-4.00$ vtcc -vt:cc mpicc -g -c prog.c
bash-4.00$ vtcc -vt:cc mpicc -g -o prog.exe prog.o
```

For others languages you have :

- *vtcc -vt:cc mpicc* : MPI C compiler
- *vtc++ -vt:cxx mpic++*, *vtCC -vt:cxx mpiCC* et *vtcxx -vt:cxx mpicxx* : MPI C++ compilers
- *vtf77 -vt:f77 mpif77* et *vtf90 -vt:f90 mpif90* : MPI Fortran compilers

By default, VampirTrace wrappers use Intel compilers. To change for another compiler, you can use the same method for MPI:

```
bash-4.00$ vtcc -vt:cc gcc -O2 -c prog.c
bash-4.00$ vtcc -vt:cc gcc -O2 -o prog.exe prog.o
```

To profile an OpenMP or a hybrid OpenMP/MPI application, you should add the corresponding OpenMP option for the compiler:

```
bash-4.00$ vtcc -openmp -O2 -c prog.c
bash-4.00$ vtcc -openmp -O2 -o prog.exe prog.o
```

Then you can submit your job. Here is an example of submission script:

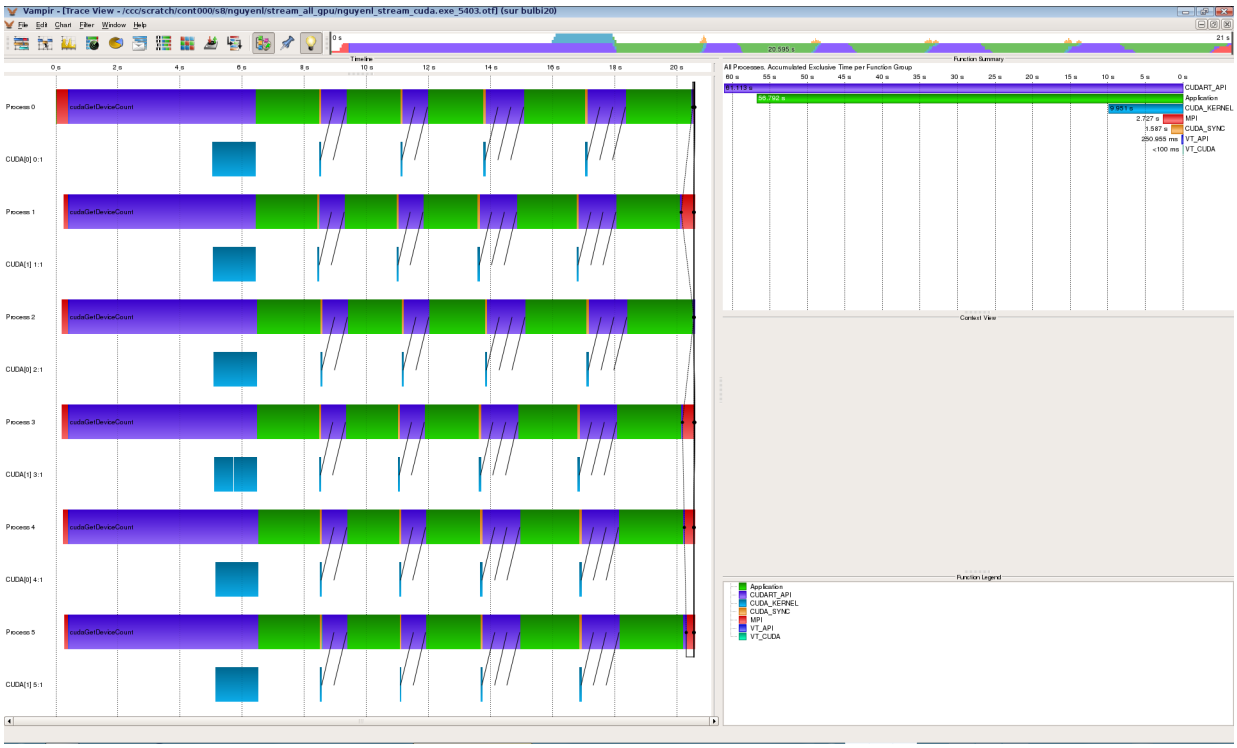
```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -e example_%I.e        # Error output. %I is the jobid


set -x
cd ${BRIDGE_MSUB_PWD}


ccc_mprun ./prog.exe
```

At the end of execution, the program generates many profiling files :

```
bash-4.00$ ls
a.out a.out.0.def.z a.out.1.events.z ... a.out.otf
```

To visualize those files, you must load the vampir module:

```
bash-4.00$ module load vampir
bash-4.00$ vampir a.out.otf
```



Vampir window

If you need more information, you can contact us.

Tips

Vampirtrace allocate a buffer to store its profiling information. If the buffer is full, Vampirtrace will flush the buffer on disk. By default, the size of this buffer is 32MB per process and the maximum number of flushes is only one time. You can increase (or reduce) the size of the buffer: your code will also use more memory. To change the size, you have to initialize an environment variable :

```
export VT_BUFFER_SIZE=64M
ccc_mprun ./prog.exe
```

In this example, the buffer is set to 64 MB. We can increase the maximum number of flushes:

```
export VT_MAX_FLUSHES=10
ccc_mprun ./prog.exe
```

If the value for *VT_MAX_FLUSHES* is 0, the number of flushes is unlimited.

By default, Vampirtrace will first store profiling information in a local directory (/tmp) of process. These files can be very large and fill the directory. You have to change this local directory with another location:

```
export VT_PFORM_LDIR=$SCRATCHDIR
```

There are more Vampirtrace variables which can be used. See User Manual for more precisions.

Vampirserver

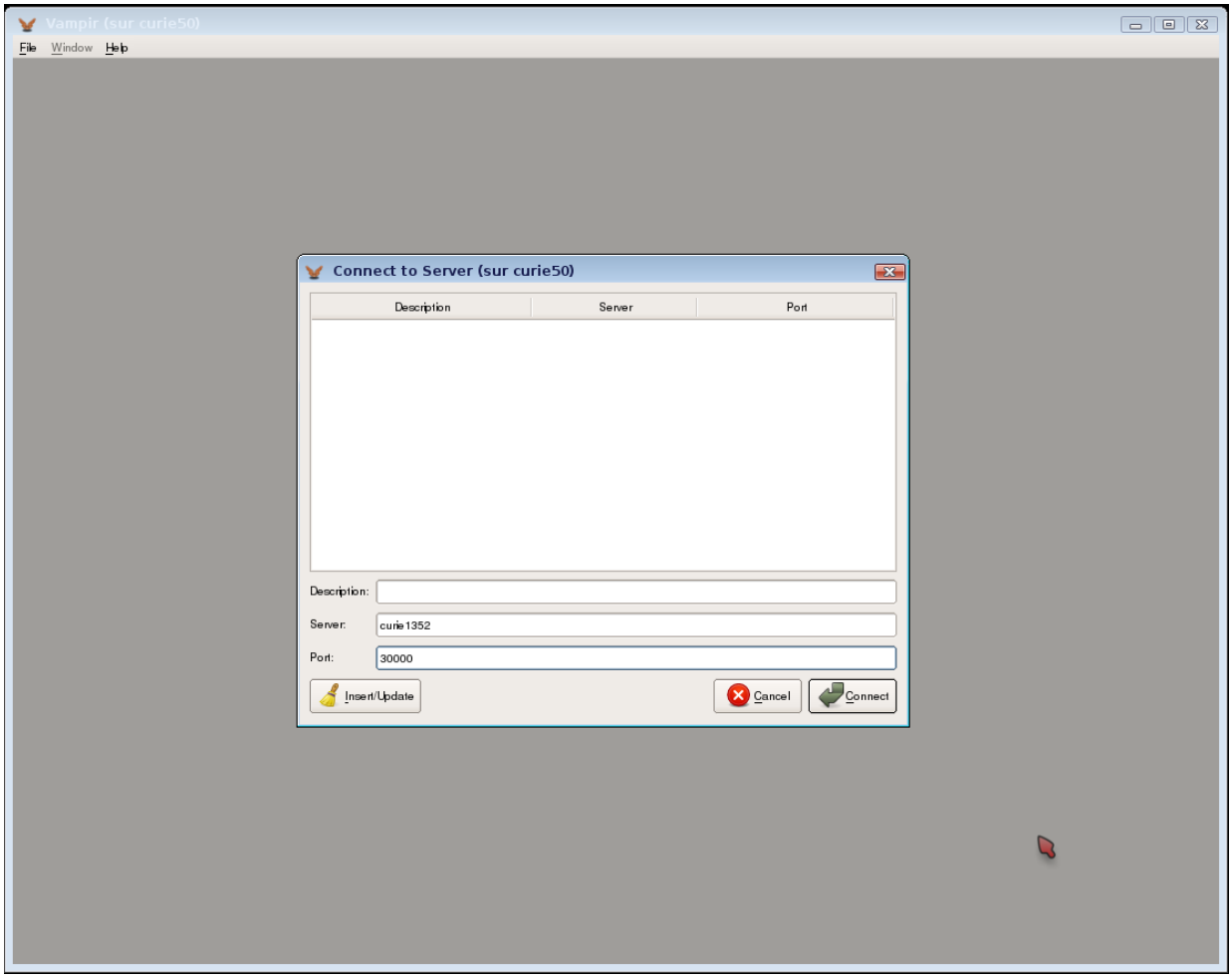
Traces generated by Vampirtrace can be very large: Vampir can be very slow if you want to visualize these traces. Vampir provides Vampirserver: it is a parallel program which uses CPU computing to accelerate Vampir visualization. Firstly, you have to submit a job which will launch Vampirserver on Curie nodes:

```
$ cat vampirserver.sh
#!/bin/bash
#MSUB -r vampirserver              # Request name
#MSUB -n 32                        # Number of tasks to use
#MSUB -T 1800                      # Elapsed time limit in seconds
#MSUB -o vampirserver_%I.o         # Standard output. %I is the jobid
#MSUB -e vampirserver_%I.e         # Error output. %I is the jobid

ccc_mprun vngd

$ module load vampir
$ ccc_msub vampirserver.sh
```

When the job is running, you will obtain this ouput:

```
$ ccc_mpp
USER       ACCOUNT    BATCHID  NCPU   QUEUE     PRIORITY   STATE  RUN   RUN/START     SUSP   OLD    NAME           NODES
toto       genXXX     234481   32     large     210332     RUN    30.0m  1.3m          -      1.3m   vampirserver   curie1352
$ ccc_mpeek 234481
Found license file : /usr/local/vampir-7.3/bin/lic.dat
Running 31 analysis processes... (abort with Ctrl-C or vngd-shutdown)
Server listens on: curie1352:30000
```

In our example, the Vampirserver master node is on curie1352. The port to connect is 30000. Then you can launch Vampir on front node. Instead of clicking on *Open*, you will click on *Remote Open*:



Connecting to Vampirserver

Fill the server and the port. You will be connected to vampirserver. Then you can open an OTF files and visualize it.

Notes:

- You can ask any number of processors you want: it will be faster if your profiling files are big. But be careful, it consumes your computing times.
- Don't forget to delete the Vampirserver job after your analyze.

CUDA profiling

Vampirtrace can collect profiling data from CUDA programs. As previously, you have to replace compilers by Vampirtrace wrappers. NVCC compiler should be replaced by *vtnvcc*. Then, when you run your program, you have to set an environment variable:

```
export VT_CUDARTTRACE=yes
ccc_mprun ./prog.exe
```

Scalasca

Scalasca is a set of software which let you profile your parallel code by taking traces during the execution of the program. This software is a kind of parallel gprof with more information. We present here an introduction of Scalasca.

Standard utilization

First, you must compile your code by adding Scalasca tool before your call of the compiler. In order to use Scalasca, you need to load the scalasca module:

```
bash-4.00$ module load scalasca
bash-4.00$ scalasca -instrument mpicc -c prog.c
bash-4.00$ scalasca -instrument mpicc -o prog.exe prog.o
```

or for Fortran :

```
bash-4.00$ module load scalasca
bash-4.00$ scalasca -instrument mpif90 -c prog.f90
bash-4.00$ scalasca -instrument mpif90 -o prog.exe prog.o
```

You can compile for OpenMP programs:

```
bash-4.00$ scalasca -instrument ifort -openmp -c prog.f90
bash-4.00$ scalasca -instrument ifort -openmp -o prog.exe prog.o
```

You can profile hybrid programs:

```
bash-4.00$ scalasca -instrument mpif90 -openmp -O3 -c prog.f90
bash-4.00$ scalasca -instrument mpif90 -openmp -O3 -o prog.exe prog.o
```

Then you can submit your job. Here is an example of submission script:

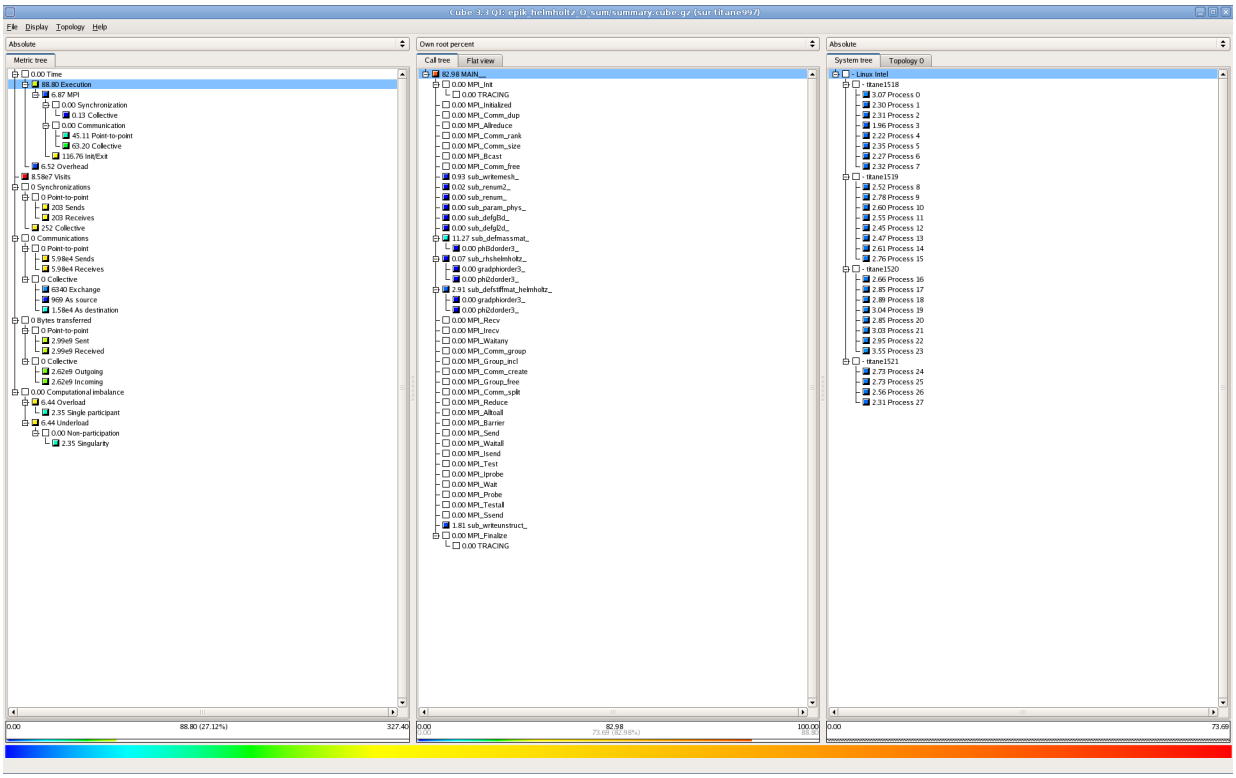
```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the job id
#MSUB -e example_%I.e        # Error output. %I is the job id


set -x
cd ${BRIDGE_MSUB_PWD}


export SCAN_MPI_LAUNCHER=ccc_mprun
scalasca -analyze ccc_mprun ./prog.exe
```

At the end of execution, the program generates a directory which contains the profiling files :

```
bash-4.00$ ls epik_*
...
```

To visualize those files, you can type:

```
bash-4.00$ scalasca -examine epik_*
```



Scalasca

If you need more information, you can contact us.

Scalasca + Vampir

Scalasca can generate OTF tracefile in order visualize it with Vampir. To activate traces, you can add *-t* option to *scalasca* when you launch the run. Here is the previous modified script:

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the job id
#MSUB -e example_%I.e        # Error output. %I is the job id


set -x
cd ${BRIDGE_MSUB_PWD}
```

```
scalasca -analyze -t mpirun ./prog.exe
```

At the end of execution, the program generates a directory which contains the profiling files :

```
bash-4.00$ ls epik_*
...
```

To visualize those files, you can visualize them as previously. To generate the OTF trace files, you can type:

```
bash-4.00$ ls epik_*
bash-4.00$ elg2otf epik_*
```

It will generate an OTF file under the epik_* directory. To visualize it, you can load Vampir:

```
bash-4.00$ module load vampir
bash-4.00$ vampir epik_*/a.otf
```

Scalasca + PAPI

Scalasca can retrieve the hardware counter with PAPI. For example, if you want retrieve the number of floating point operations :

```
#!/bin/bash
#MSUB -r MyJob_Para          # Request name
#MSUB -n 32                  # Number of tasks to use
#MSUB -T 1800                # Elapsed time limit in seconds
#MSUB -o example_%I.o        # Standard output. %I is the jobid
#MSUB -e example_%I.e        # Error output. %I is the jobid


set -x
cd ${BRIDGE_MSUB_PWD}


export EPK_METRICS=PAPI_FP_OPS
scalasca –analyze mpirun ./prog.exe
```

Then the number of floating point operations will appear on the profile when you visualize it. You can retrieve only 3 hardware counters at the same time on Curie. The the syntax is:

```
export EPK_METRICS="PAPI_FP_OPS:PAPI_TOT_CYC"
```

Paraver

Paraver is a flexible performance visualization and analysis tool that can be used to analyze MPI, OpenMP, MPI+OpenMP, hardware counters profile, Operating system activity and many other things you may think of!

In order to use Paraver tools, you need to load the paraver module:

```
bash-4.00$ module load paraver
bash-4.00$ module show paraver
-------------------------------------------------------------------
/usr/local/ccc_users_env/modules/development/paraver/4.1.1:

module-whatis   Paraver
conflict  paraver
prepend-path  PATH /usr/local/paraver-4.1.1/bin
prepend-path  PATH /usr/local/extrae-2.1.1/bin
prepend-path  LD_LIBRARY_PATH /usr/local/paraver-4.1.1/lib
prepend-path  LD_LIBRARY_PATH /usr/local/extrae-2.1.1/lib
module    load papi
setenv    PARAVER_HOME /usr/local/paraver-4.1.1
setenv    EXTRAE_HOME /usr/local/extrae-2.1.1
setenv    EXTRAE_LIB_DIR /usr/local/extrae-2.1.1/lib
setenv    MPI_TRACE_LIBS  /usr/local/extrae-2.1.1/lib/libompitrace.so
-------------------------------------------------------------------
```

Trace generation

The simpliest way to activate mpi instrumentation of your code is to dynamically load the library before execution. This can be done by adding the following line to your submission script:

```
export LD_PRELOAD=$LD_PRELOAD:$MPI_TRACE_LIBS
```

The instrumentation process is managed by Extrae and also need a configuration file in xml format. You will have to add next line to your submission script.

```
export EXTRAE_CONFIG_FILE=./extrae_config_file.xml
```

All detailled about how to write a config file are available in Extrae's manual which you can reach at $EXTRAE_HOME/doc/user-guide.pdf. You will also find many examples of scripts in $EXTRAE_HOME/examples/LINUX file tree.

You can also add some manual instrumentation in your code to add some specific user event. This is mandatory if you want to see your own functions in Paraver timelines.

If trace generation succeed during computation, you'll find a directory *set-0* containing some *.mpit* files in your working directory. You will also find a *TRACE.mpits* file which lists all these files.

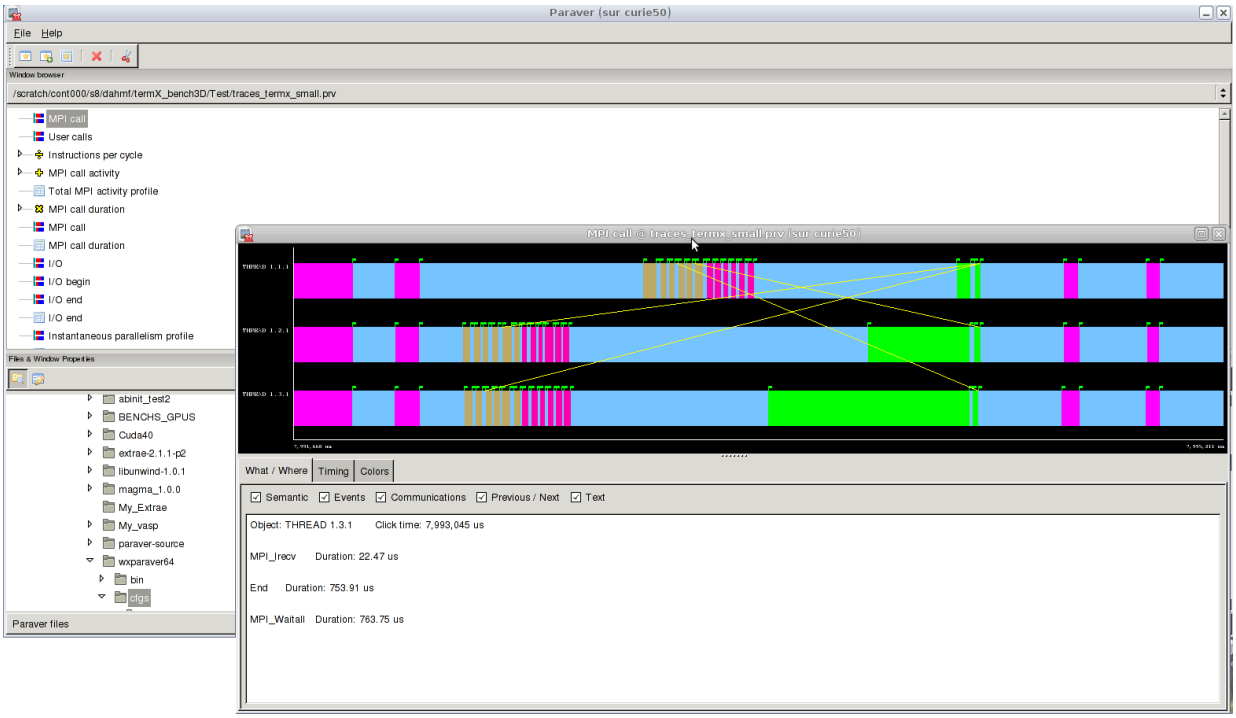
Converting traces to Paraver format

Extrae provides a tool named mpi2prv to convert mpit files into a .prv which will be read by Paraver. Since it can be a long operation, we recommend you to use the parallel version of this tool, mpimpi2prv. You will need less processes than previously used to compute. An example script is provided below:

```
bash-4.00$ cat re build.sh
#MSUB -r merge
#MSUB -n 8
#MSUB -T 1800

set -x
cd $BRIDGE_MSUB_PWD
ccc_mprun mpimpi2prv -syn -e path_to_your_binary -f TRACE.mpits -o file_to_be_analysed.prv
```

Launching Paraver

You just now have to launch "paraver file_to_be_analysed.prv". As Paraver may ask for high memory & CPU usage, it may be better to launch it through a submission script (do not forget then to activate the -X option in ccc_msub). For analyzing your data you will need some configurations files available in Paraver's browser under *$PARAVER_HOME/cfgs* directory.



Paraver window